Shift To Swift

Programming Ios Apps

By: Mofeed Ali Al-shouib

مسموح فقط للنشر الإلكتوني

العادات الخاطئة في تعلم البرمجة!

في مقدمة هذا المقال الذي سوف نتاول فيه موضوع مهما جداً قبل البدئ في مشروع البرمجة و خوض المعركة فيها . ألا و هو العادات الخاطئة في التعلم إن صح التعبير و ليس حصراً على البرمجة. مثلما كنت في ذاتِ يوم شغوفا في قراءة المقالات منذ الصغر ، أي حوالي قبل ٩ أو أكثر من ذلك بقليل حول البرمجة لم أنجح إلا في السنة الأخيرة هذه لعدة عوامل ، و هي ما سوف نناقشه ، أيضاً في هذه المقالة.

هل أنت شغوفاً للبرمجة حقاً؟ ذلك السؤال الذي ترسخ في دماغي بعد المحاولات في السنين الماضيه ، وجدت نفسي شغوفا للبرمجة و لست شغوفا لفهمها . البرمجة كأي لغة تريد تعلمها ، لها أساسياتها ، مصطلحاتها ، و طريقة التعبير عنها .

لن أصمم إلا برنامج إحترافي! في هذا التعبير سأضرب عصفوران بحجرٍ و احد . كثير ممن يودون سلوك طريقة البرمجة يعتقدون بأن البرمجة تساوي التصميم . و هذا جدا من الخطأ لأن المصمم وظيفة ذوقية و البرمجة و ظيفة مهنية و تضلع في اللغة المراد تعلمها . و أما عن العصفور الآخر ، فهو الإحتراف من نقطة البداية . و هذا التفكير كنت آخذ به بعين الإعتبار مما أدى إلى شعوري بالملل و من ثم تحاشي البرمجة لمدة طويلة.

في يوم سوف أتعلم! لا ندرك مثل هذه الأخطاء في زمن السرعة . فقد أكْسَبَنا صفة السرعة معه . بحيث نسينا من ذي قبل كم نقع لنخطو مع مرور السنين ، و كم حاولنا قول "ماما" و "بابا" في أول السنوات. و جدنا الصعوبة في البداية و لكن بعد مضي الشهور و السنوات أصبحنا أكفاء ، فنطقنا بأخي و أختي و أعطني اللعبة. كما إنتقلنا لمرحلة أخرى بعد إكمالنا للنطق و هي الإدراك و التمييز. لذلك العملية العلمية تحتاج إلى وقت لبزوغ فهمها بالشكل السليم.

الوقت لا يكفي! هل شعرت يوماً بأن الوقت لا يكفي للتعلم! و أنشأت جدولاً فيه جميع أوقاتك فاكتشفت أنّ الروتين ممل! هل أدركت أنّ النفس البشرية تقمع الروتين! ما هو الشيء الأنسب كي أجعل توافقاً بين البحث عن الهدف و العمل أو الدراسة و المنزل!؟ لعل من أبرز أسباب العزوف أيضا هو الظن بأنك تستطيع إدارة الوقت ، ذلك وهم لا يمكنك إدارة الظروف إن جاءت لذلك سوف تقول غدا أبداً ويأتي الغد و يصطحب ظرفاً آخر هكذا دواليك. لذلك خصص لك ساعة يوميا أو نصف الساعة ما سكاً أحد المواضيع التي تخص هدفك.

لا توجد فائدة للتدوين! الإنسان يستقبل المعلومة إما عن طريق السمع أو القراءة ، و تخزينها في الدماغ عن طريق النظر كقراءة كتاَب أو مشاهدة الفيديو أو عن طريق السمع كسماع صوتية أو من خلال تكرار المعلومة شفهياً. لذلك من أفضل الطرق لتخزين المعلومات التدوين فلها ميزات الثلاث تستطيع رؤيتها و تكرارها بنفسك و سماع نفسك أثناء تكرارها ، فهي من الطرق الناجحة التي أستطيع أن أقول عنها غيرت من مجرى فهمي في لغة البرمجة.

الإنجليزية صعبة ما الحل! البرمجة ليست إنجليزية معقده أبداً ، هل تعلم ذلك! هل يوما ما استخدمت كلمة أو عدة كلمات من لغة ثانية للتعبير عنها مُزحة مع صديقٍ لك! أو زميل عمل ؟ مثل "كمستكا" باللغة الفلبينية و تعني مرحبا ، هل احتجت تعلم قواعد اللغة الفلبينية كي تنطق "كمستكا"! كذلك هو الحال في البرمجة لا تحتاج منك تلك الخبرة بجميع اللغة الإنجليزية . لذلك الإعتقاد بأنها حاجز وهم. لذلك أنت تحتاج فهم الكلمات التي يستدعيها المترجم.

ما هي لغات البرمجة ؟

إنّ البرنامج في الحاسوب أو الأجهزة الحديثة تقوم بتنفيذ ما يأمرها المبرمج بأحدى لغات البرمجة مثل اللغة **swift** و مثل هذه اللغات يطلق عليها اللغات البرمجية العليا و من ثم تحولها إلى لغة الآلة. وكذلك يجب أن لا نتغافل عن أنّ البرنامج عبارة عن مجموعة متسلسلة من الأوامر.

ماهي لغة الآلة ؟ لغة الآلة (**Machine Language**) أو ما تسمى (**اللغة الثنائية**) . هي اللغة الوحيدة التي يفهمها الجزء الداخلي ، و تتكون من سلسلة من الرقمين 0 و 1 ، وتعتبر من أصعب لغات البرمجة لسبب كون جميع الأمور تتنفذ من سلسة الصفر و الواحد وأقل خطأ سينتج عنه خلل.

ما هي اللغات العليا! اللغات العليا هي تلك اللغات التي يستطيع البشر فهمها بشكل أوضح و التي تتمثل من خلال الكتابة للأوامر باللغة الإنجليزية مثل لغة **swift** و **java** وغيرها من اللغات . بحيث تتميز هذه اللغات بعدم الزام المبرمج معرفة التفاصيل الدقيقة لعمل الحاسب. كما تحتوي على **compiler** (مترجم) الذي من شأنه تحويل الكلام المكتوب بلغة البرمجة إلى لغة الآلة.

مقدمة عن لغة سويفت

لغة swift لغة حديثة العهد فقد أصدرتها أبل في عام ٢٠١٤ م ، و التي كانت بمثابة المفاجأة للجميع . بحيث تتميز عن اللغة التي استخدمتها أبل في الـ ٢٠ عام الماضية و هي **Obj-c** في السرعة و الأمان ، وسوف نستعرض الفوارق بين السويفت و الاوبجكتف سي في نقاط.

السريعة و القوية! نعم فلقد صممت لغة سويفت لتكون قوية و سريعة . فعلى سبيل المثال قد تختصر لك كود بسطر واحد بدلا من خمسة أسطر في **obj-c** . وأما من ناحية القوى فتعتبر قوية لكونها موروثة من **C** و **Obj-C** .

الأكثر سهولة! من مميزات اللغة أنها بالفعل صديقة للمبتدئين ، بحيث تعتبر اللغة الأكثر صعوبة بين لغات البرمجة هي **Obj-C** و جاءت لغة **swift** لتحل مشكلة الصعوبة. فالأسطر تبدو أكثر وضوحاً و أسهل في القراءة و الفهم.

لغة آمنة! تمتاز **Swift** بأمان فائق فهي تتعرف على الأخطاء الموجودة في التعليمات البرمجية التي قد تسبب خلل في التطبيقات أو نظام الجهاز و تُزيلها وتقوم بإدارة الذاكرة تلقائياً لتعزيز نسبة الأمان. كما أن سهولة و وضوح كتابة التعليمات البرمجية جعل من السهل تمييز عملها و تحديد مهمتها أو الغرض من وجودها.

تطبيق الكود! توجد ميزة جديدة مع اللغة الحديثة تسمّى **"PlayGround"** ، وظيفتها تسهل عليك تجربة الكود ، دون الدخول في برمجة الواجهآت.

برمجة كائنية التوجه

البرمجة كائنية التوجه أو ما تُسمى بـ Object Oriented Programming هي النمط المستخدم حاليا في غالبية لغات البرمجة و التي تعتمد على الكائنات في البرمجة فكل شيء في الحياة عبارة عن كائن و كل كائن له خصائص **properties** و أفعال **Actions** و سوف نأخذ السيارة ككائن و نعطيه الخصاص و الأفعال في الجدول التالي.

خصائص	أفعال
اللون	كيف تعمل السرعة
الشكل	كيفية الالتفاف
تاريخ الصنع	التوقف عند ضغط المكابح
الشركة المصنعه	توقف الإطارات عند رفع الهند بريك

كما يرتبط تماما مع موضوع البرمجة كائنية التوجه أمور الوراثة و الفئات و التعدد الصوري. و فيما يلي من فصول سوف نناقش جميع هذه المواضيع على حذا .

تحميل xcode

في البداية يجب التعريف ببرنامج الـ xcode و هو المنصة التي يعتمد عليها المبرمج في إنشاء التطبيقات ، والذي بـــدوره يـــحتوي على الأدوات التي أعدتها أبل مسبقا و في الـغالـب تـتواجـد في الـنظام الأسـاسـي كـالأزرار و كقائـمة الإعـدادات . كـمـا تــرى في الـصورة الآتـية فهذه الـواجـهة التي تظهر عندما تضغط على إيقونة الـبرنـامج. و للملاحـظة فـالمسـتطيل الأزرق الـفاتح ع اليمين يحـتوي على المشـاريع التي عملتها سـابقا. بحـث لن يظـهر أي شـيء في المرة الأولى من فتحك للبرنامج. وأما عن الخيارات الثلاث في اليسار فالأول منهم لإنشاء ملف PlayGround و الذي سوف نتعلم فيه جميع أسـاسـيات البرمجة ، و الـخيار الثاني فهو لإنشاء تطبيق جديد و هذا الزر سنستخدمه في ما بعد الأساسيات.

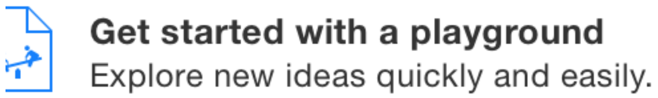
Pic 1

كيف أحصل على xcode ؟ للحصول على برنامج الاكس كود 6 و هو المعني بلغة **swift** لابد من الحصول على حساب مطور مجاني أو مدفوع. فالمجاني فقط يتيح لك تحميل البرنامج من الآب ستور من أي حساب مجاني على الآب ستور. و أما المدفوع ليس ضروريا في هذا الوقت لأنه معني لتحميل برنامجك الخاص على الآب ستور أو تجـربـة الـبرنـامج على جـوالـك الحقيقي. أو تسـتطيع تحـمـيل الـبرنـامج مـن الـرابـط التالي. http://developer.apple.com/xcode/downloads

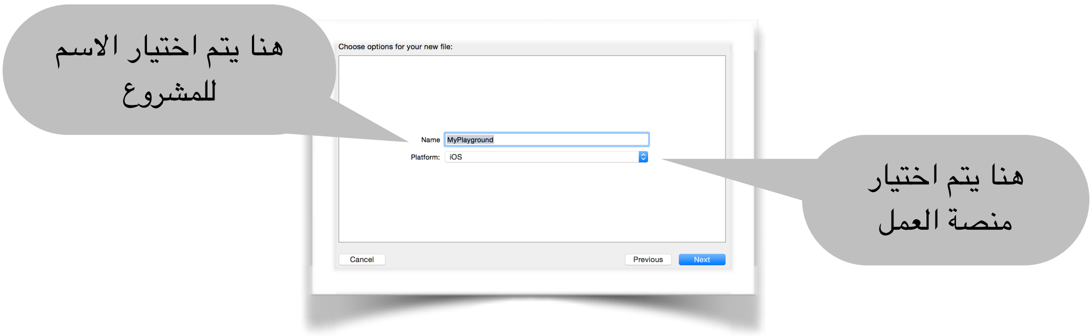
المتغيرات و الثوابت

نفتح البرنامج بالبداية ثم نتوجه إلى الخيار الأول و هو لإنشاء الـ PlayGround

ماهو الـ PlayGround! خاصية تجعلنا نستخدم الكود البرمجي وإختباره دون الولوج في برمجة الواجهة.



- نقوم بإنشاء **PlayGround** جديد
- ثـــم سنسجـــل الإســـم ونـــختار Ios إلى platform
- بعد ذلك نضغط على كلمة **Next**
- ستُفتح لك هذه الشاشة ففي يمين الشاشة نتيجة الكود المكتوب و على اليسار الكود البرمجي المراد كتابته.
- امسح الكود بشكل كامل لنبدأ الرحلة مع أول درس بالأساسيات.



ما المتغيرات و ما هي الثوابت! لكل إنسان على هذا الكوكب عندما يولد اسم خاص به يرشدنا إليه لمعرفته ، كذلك في البرمجة نحتاج لتعريف متغيرات أو ثوابت يحملون اسم واحد فقط لا يمكن تكراره ، فعلى سبيل المثال قد نعرف متغير اسمه **x** و يحمل القيمة صفر . تذكر أنّ المتغير يحمل قيمة يتم حجزها على الذاكرة.

ما الفرق بين المتغيرات و الثوابت! إنّ الفارق الكبير و الواسع بينهما جذري جداً. فالمتغيرات هي عبارة عن قيمة يمكن تغييرها داخل الكود البرمجي بحيث لو وُضعت القيمة الأولية لمتغير اسمه **x** و تساوي عشرة تستطيع مستقبلا داخل الكود البرمجي أن تُغيره. أما الثوابت فعلى النقيض تماما عندما تُعطيها قيمة أولية و تُريد مستقبلا تغييرها تظهر لك رسالة على الشاشة تُخبرك بخطأ ما.

كيف نُصرح عن المتغيرات و الثوابت! في لغة سويفت نحتاج لكتابة متغير الكلمة الآتية **Var** وهي اختصار إلى الكلمة **Variable** كما نحتاج لتعريف الثابت كلمة **let** وهي اختصار لـ **constant** ، انظر في المثال التالي.

```
var x = 1

let c = 0
```

هل هناك عدة طُرق لتعريف المتغيرات! نعم توجد طريقتان لتعريف المتغيرات . طريقة تسمى التعريف الصريح و الثانية التعريف الغير صريح. انظر للمثال الآتي.

Y

لاحظ في التعريف الصريح خاطبنا الـ المترجم compiler أن يقوم بتحديد النوع ، لكن من احدى مميزات لغة سويفت التي تجعلها الأكثر سهولة أنّه لا نحتاج لتحديد نوع المتغير فهو ذكي بعد إعطائه القيمة الأولية

```
var x:Int = 1
```

يبحث عن نوعه. و فيما يتقدم سوف نطرح أنواع المتغيرات في جدول.

المتغيرات الإختيارية! لا تقلق بشأن هذا المصطلح الجديد . فهو سهل ، بحيث أننا نقوم بتعريف متغير لا يحمل قيمة ، بحيث أنه من الممكن أن يكون فارغا "**nil**". انظر للمثال الآتي. و لاتنسى فعلامة الإستفهام هنا للتعبير عن الخاصية الإختيارية.

أنواع المتغيرات! بطبيعة الحال القيم في البرمجة تختلف كما في حياتنا فتارة نكتب قيمة رقمية و تارة نص و تارة عدد عشري . تابع الجدول التالي لفهم ذلك.

```
var x:Int?      nil
```

النوع	المثال	طريقة الاستدعاء
Integer	**5,10-,2**	**var x:Int = 1**
Float	**2.4 , 10.0 , 0.5**	**var x:Float = 1.0**
Double	**4.3212- , 2.334**	**var x:Double = 1.435**
Boolen	**True , False**	**var x:Bool = True**
String	أي نص	**var x:String= "xcode"**

ملاحظة : النوع Boolen لا يحمل فقط إلا قيمتان هما True , False

ملاحظة : النوع String لابد للقيمة المراد ذكرها وضعها بين علامة التنصيص ""

ما الطريقة المُثلى لكتابة أسماء المتغيرات!

- إبدأ بكتابة الكلامة الأولى بالأحرف الصغيرة ثم من كل كلمة أخرى اجعل الحرف الأول كبيرا مثل numberOfStudent
- لا تكتب الحرف الأولي رقما
- لا تكتب اسم كلمة محجوزة للمترجم compiler

التعليقات

لماذا التعليقات! قد لا تُدرك الآن أهمية التعليقات في الكود البرمجي لكونك لم تعمل على برنامج ضخم . لذلك أود تنبيهك لأهميتها مستقبلا . في حال أردت الرجوع مستقبلا للبرنامج و إظافة بعض التعديلات عليه أو أردت أن تُشارك الأصدقاء برنامجك ليجعل منه أكثر فهما . بحيث أنّ هذه الخدمة تتيح لك فرصة كتابة التعليقات داخل الكود البرمجي دون احداث تلف للبرنامج.

كيف أكتب التعليق! نكتب التعليقات باستخدام backslash مضاعف "//" و من ثم نرفق التعليق المراد تعليقه. انظر للمثال الآتي.

```
// here write anything
```

هذه الطريقة ناجحة! الطريقة السابقة ناجحة لكن لسطر واحد فقط ، أي في حال أنك ضغطت على الزر انتر وأردت إتمام التعليق فلن يكون تعليقا ، إذا ما الحل ! اتبع الصورة الآتية.

نبدأ بوضع */ و ننهي بوضع /*

```
/*

اكتب ما يحلو لك
في هذا المكان

*/
```

العمليات الحسابية

ما هي العمليات الحسابية! العمليات الحسابية في لغات البرمجة هي كالمتعارف عليها عندنا ، مثل الضرب و القسمة و الجمع و الطرح ، و لكن الإختلاف الجوهري فقط يختلف بالرمز الذي يُعبر عن كل عملية. ففي الواقع نستخدم x اشارة للضرب بينما في البرمجة نستخدم * للتعبير عن الضرب.

نوع العملية	رمز التعبير عنها
الضرب	*
القسمة	/
الطرح	-
الجمع	+
باقي القسمة	٪

هل يمكنني الإختصار! تستطيع اختصار بعض العمليات . انظر للجدول الآتي.

الوصف	التطبيق
أضف قيمة الـ x إلى y و أعد تخزين القيمة لـ x	x + = y
أنقص قيمة الـ x من y و أعد تخزين القيمة لـ x	x - = y
أضرب قيمة الـ x في y و أعد تخزين القيمة لـ x	x * = y
أقسم قيمة الـ x على y و أعد تخزين القيمة لـ x	x / = y
خذ باقي القسمة من x على y وخزنها في الـ x	x % = y
تستخدم في الشروط و تعني "و"	x && y
تستخدم في الشروط و تعني "أو"	x ll y

Tuples

ماهو التوبلز! لا تقلق عندما تسمع بهذا المصطلح فقط بالرجوع للمعنى العام في اللغة الانجيزية سوف تفهم ما ذا يُشير إليه هذه الكلمة. فإنّ المعنى لها هو المجموعات . و الفكرة من هذه أنّك تستطيع أن تكتب متغير يشير مثلا إلى السيارة "مورسيدس" و تضع داخل هذا المتغير عدة قيم مختلفة النوع. انظر للمثال الآتي.

```
var car = ("morsidas" , "black" , 2015)
car.0
car.1
car.2
```

```
(.0 "morsidas", .1 "black", .2 2015)
"morsidas"
"black"
2015
```

في المثال السابق يتضح المثال ، فبدل من أعرف ثلاثة متغيرات للسيارة قمت بتعريف متغير واحد و يحمل عدة قيم مختلفة النوع . وهذه هي ميزة التوبلز.

كيف أستدعي التوبلز! لاستدعاء قيمة داخل التوبلز نحتاج لمعرفة الاسم للمتغير الذي عرفناه و في مثالنا هو car و بعد ذلك نُلحق الاسم بـ النقطة " . " و من ثم ستظهر لك قيم من الرقم صفر تبدأ و تنتهي بعدد القيم التي وضعتها . و في مثالنا و ضعنا ثلاث قيم . لذلك انتبه جيداً أنّك هنا آجباري تبدأ من الرقم صفر و على هذا يترتب بأن القيمة **الأولى** ستساوي **صفر** عند الاستدعاء و القيمة رقم **اثنين** ستساوي **واحد** عند الاستدعاء و كذلك القيمة **الثالثة** ستساوي رقم **اثنين** عند الإستدعاء.

هل الطريقة السابقة ناجحة! لا نستطيع أن نقول تلك الطريقة غير ناجحة لكننا نستطيع أن نقول و نجزم بوجود طريقة أخرى أكثر فاعلية من هذه. بحيث في هذا المثال القادم سوف نوضح طريقة تجعلنا استبدال الأرقام عند الاستدعاء إلى نص يشير للقيمة ، فمثلا في القيمة الأخيرة وهي اللون عندما استدعيناها كان الرقم ٢ . و أما الآن سوف نضع بدل الرقم نص و يكون color . تابع المثال.

```
var car = (name:"morsidas" , color:"black" , year:
    2015)
car.color
car.name
car.year
```

```
(.0 "morsidas", .1 "black", .2 2015)

"black"
"morsidas"
2015
```

نلاحظ أننا قبل القيمة وضعنا اسم ثم وضعنا بعده النقطتان الرأسيتان " : "

المصفوفات Array

ما هي المصفوفات! المصفوفات ببساطة عبارة عن متغير من نوع مصفوفة ، يحوي بداخله مجموعة من القيم ، و لكن يشترط أن تكون هذه القيم من النوع نفسه ، بحيث أنّ عندما تُنشئ مصفوفة من نوع **Int** لابد أن تكون كل القيم من هذا النوع .

لماذا المصفوفات! المصفوفات من الأشاء المهمة عندما تبدأ برنامجك الكبير . فعلى سبيل المثال قد تقوم بإنشاء مصفوفة تحوي على قيمة من نوع **String** و التي بدورها تحوي أسماء مجموعة من الصور التي تود مثلا عرضها على الشاشة للمستخدم. أو مثلا مصفوفة تحتوي على مجموعة من الأرقام وتريد استخدامها أكثر من مرة في البرنامج فهي تغنيك عن اعادة الكتابة للمعلومات مرة ثانية.

أنواع المصفوفات! المصفوفات نوعين ، المصفوفات الثابتة ، و المصفوفات المتغيرة. فالثابته هي المصفوفة التي تحمل قيم معينة لا يُمكن تغييرها أبدا خلال مسار البرنامج. و أما المصفوفات المتغيره فهي التي يمكن تغير قيمتها في مسار البدنامج.

كيف أُنشئ مصفوفة! انظر للمثالين الآتيين. فالأول يمثل مصفوفة متغيره. و أما الثاني فمصفوفة ثابتة.

```
var array = ["ali" , "Ahmad"]
let array = ["ali" , "Ahmad"]
```

المصفوفات الفارغة! قد تحتاج لإنشاء مصفوفة فارغة مبدئاً ثم من خلال مسيرة البرنامج تعمل على إضافة قيمة لهذه المصفوفة. في المثال القادم سوف نتعلم كيف ننشئ مصفوفة فارغة و تحديد نوع القيمة المراد تخزينها فيه.

```
var array = [String]()
```

بعض الدوال المهمة! سنتحدث عن بعض الدوال المهمة للمصفوفات. لا تقلق بشأن كلمة دوال إن لم تفهمها الآن. كل ما عليك فهم وظيفة الدالة في هذا الدرس.

دالة معرفة حدود المصفوفة! المقصود بها عدد العناصر المتواجدة داخل المصفوفة و اسم الدالة **count** و تعني بالعربية عد. انظر للمثال التالي.

نلاحظ بعدما استدعينا اسم المصفوفة و ضعنا النقطة ثم استدعينا الدالة الخاصة بعمل العد للعناصر الداخلة في المصفوفة.

```
var array = [String]()     0 elements
array.count                0
```

دالة الإضافة! قد تحتاج لإظافة عناصر جديدة لمصفوفتك . لذلك يجب معرفة الدالة الخاصة بذلك و هي **append** . ملاحظة هذه الدالة تضيف القيمة لنهاية المصفوفة.

```
var array = [String]()       0 elements
array.append("mofeed")       ["mofeed"]
```

دالة الإضافة الثانية! في هذه الدالة الأخرى مميزة . تجعلك تختار موضع القيمة التي تريدها في المكان الذي تريده . بحيث أنّ المكان رقم **واحد** يكون عند **Index 0** كما المكان رقم **اثنان** يكون عند **index 1** و هكذا. لا تقلق بشأن الـ **index** ستفهمه عما قريب.
انظر للمثال الذي يشرح دالة الـ **insert** .

```
var array = [String]()                  0 elements
array.insert("mofeed", atIndex: 0)      ["mofeed"]
```

دالة الحذف! كما هنالك ميزة تمكنك من الاضافة هنالك ميزة تمكنك من الحذف. و التي تسمى **removeAtIndex** . انظر للمثال التالي.

```
var array = [String]()                  0 elements

array.insert("mofeed", atIndex: 0)      ["mofeed"]
array.removeAtIndex(0)                  "mofeed"
array.count                             0
```

القاموس Dictionary

ماهو القاموس! نوع من أنواع تخزين القيمة بقيمة أخرى. فعلى سبيل المثال نستطيع تخزين كلمة إنجليزية بمعناها العربي. و تذكر أيضاً أنه يتكون من مفتاح **kay** و من قيمة **value** . انظر للشرح الآتي لتفهم كيف يعمل القاموس. بحيث يتم تعيين لكل مفتاح قيمة.

كيف تتم كتابة القاموس! تتم كتابة القاموس بعد تعريف متغير و تحديد نوع قيمة المفتاح و تحديد نوع القيمة الآخرى. انظر للمثال التالي.

في هذا المثال البسيط قمنا بتعريف قاموس يحمل قيمتين و مفتاحين .

```
var dic = ["ok":"yes" , "no":"Not"]      ["no": "Not", "ok": "yes"]
// استدعاء
dic["ok"]                                "yes"
```

معرفة عدد العناصر ! لمعرفة عدد العناصر داخل القاموس نستخدم الدالة التالية **Count** و هي المعنية بحسب العناصر. انظر للمثال التالي.

```
dic.count      2
```

إضافة عنصر جديد! نستطيع إضافة عنصر جديد و لكن ليس من خلال دالة في القاموس كما هو الحال في المصفوفات . انظر للمثال الآتي.

```
var dic = ["ok":"yes" , "no":"Not"]      ["no": "Not", "ok": "yes"]
dic["mofeed"] = "mof"                    "mof"
dic                                      ["no": "Not", "mofeed": "mof", "ok": "yes"]
```

لاحظ أنّ المفتاح هو الكتاب الذي بين [ ] ، و أما القيمة فهي التي بعد علامة الـ = .

كيف يتم حذف قيمة معينة! نعم نستطيع حذف مفتاح معين من خلال دالة تُسمى **removeValueForKey** ، و التي بدورها تقوم حذف المفتاح من خلال اسمه. انظر للمثال التالي.

```
var dic = ["ok":"yes" , "no":"Not"]      ["no": "Not", "ok": "yes"]
dic.removeValueForKey("ok")              "yes"
dic                                      ["no": "Not"]
```

قاعدة الـ If

ما هي قاعدة الشروط! الشروط في جميع لغات البرمجة قاعدة ثابته و مهمه . و لكل شرط جواب و الجواب هو ما سينفذه الحاسب الآلي في حال أنّ الشرط تحقق. فعلى سبيل المثال الشرط الآتي "في حال أنّ كلمة المرور و اسم المستخدم صحيحين"، الجواب "ادخل إلى لوحة التحكم". انظر للمثال الآتي.

```
var x = 0
var y = 0

if (x == 0)
{
println("good")
}
```

```
0
0

"good"
```

في هذا المثال قمنا بتعريف متغيران x , y ثم كتبنا أداة الشرط If وهي تعني إذا ، ثمّ وضعنا الشرط المراد التحقق منه داخل القوسين (x == 0) نفذ الأمر الذي بين { println ("good") } . تذكر أن الأمر المراد تنفيذه يسمى البلوق و هو الذي يكون بين { } . كذلك الأمر println عبارة عن دالة جاهزة من قبل تم استدعاءها و وظيفتها طباعة نص على الشاشة . و نحن أردنا في حال تحقق الشرط يطبع لنا كلمة good . ، كما ترى على يمين الشاشة في المثال الذي بالأعلى تمت طباعة الكلمة.

ماذا لو لم يتحقق الشرط! في المثال السابق في حال تحقق الشرط و إذا لم يتحقق الشرط فلن يفعل أي شي لذلك جاءت اللغة البرمجية بشيء يُدعى **else** . و وظيفتها في حال لم يتحقق الشرط الأول ففعل ما يلي. انظر و قارن فيما يلي .

```
var x = 1
var y = 0

if (x == 0)
{
println("good")
}else{
    println("Not goog")
}
```

```
1
0

"Not goog"
```

لاحظ في المثال على اليمين لم يتحقق الشرط لأن قيمة الـ x تساوي ١ . لذلك تم تنفيذ البلوق الخاص بـ **else** و لم يتم تنفيذ البلوق الخاص بالشرط المكتوب. و على العكس تماما المثال الذي يتلوه على اليسار.

ماذا لو توجد شروط كثيرة! في حال وجود أكثر من شرط مراد تنفيذه سنستخدم الأمر التالي **else if** .انظر للمثال التالي.

```
var x = 1
var y = 3

if (x == 0)
{
println("good")

}
else if (y == 3){

    println("y")
}
else{
    println("Not goog")
}
```

```
1
3

"y"
```

أضفنا شرط ثاني الذي هو في حال أنّ قيمة y تساوي ٣ نفذ الامر التالي و هو طباعة الحرف y . و كما سبق علمنا أنّ البرنامج عبارة عن عدة أوامر متسلسلة لذلك لو نعود للخلف مجددا لنقرأ الكود بشكل كامل سنفهم التالي.
إذا تحقق الشرط الأول نفذ البلوق الخاص فيه.
إذا لم يتحقق إنتقل للشرط الذي يليه .
إذا الشرطان لم يتحققان نفذ البلوق الخاص بـ **else**.

هل من الممكن ربط أكثر من شرط لتحقيق جواب واحدا! نعم من الممكن ذلك باستخدام " && " و الذي يعني" و " . باستخدام " || و تعني " أو ".

```
var x = 1
var y = 3

if (x == 1 && y == 3)
{
println("good")
}
```

```
1
3

"good"
```

في هذا المثال لابد من تحقق أن الشرطان صحيحان لتنفيذ البلوق الخاص فيهما .

```
var x = 1
var y = 0

if (x == 1 || y == 3)
{
println("good")
}
```

```
1
0

"good"
```

في هذا المثال . في حال تحقق أحد من هذين الشرطين سوف ينفذ البلوق . لذلك نجد بأنه لم يتحقق الشرط الثاني لكنه نفذ البوق.

المقارنة بين قيمتين في الشرط! تابع الجدول الآتي لتفهم أوجه المقارنة .

وجه المقارنة	الرمز
<	أصغر من
>	أكبر من
...	المدى ، مثلا من الرقم ١ الى ٥
علامة يساوي مرتين	دليل على مساواة القيمتين
!=	لا تساوي

=<	أصغر من أو يساوي
=>	أكبر من أو يساوي

انظر للمثال الآتي.

```
var x = 0

if (x < 1)
{
println("good")

}
```

```
0



"good"
```

هنا تحقق الشرط و المقارنه لذلك تم تنفيذ البلوق الخاص. كما تستطيع استبدال إشارة أصغر من بأي نوع من الأنواع السابقة التي تم شرح وظيفتها في الجدول.

قاعدة الـ switch

ماهي قاعدة الـ switch ! هذه القاعدة شبيهه للقاعدة السابقة . قاعدة الشرط ، لكنها تستخدم في مواطن مختلفة و تلك في مواطن مختلفة . ففي هذه القاعدة سنقوم بتحديد متغير أو أكثر من متغير لتحديد عدة حالات قد يمر فيها. فعلى سبيل المثال مستوى الممتاز يكون من ٩٠ الى ١٠٠ ، و مستوى الجيد جدا من ٧٠ الى ٨٠ ، و مستوى الجيد من ٦٠ الى ٦٩ ، وأقل من ذلك فهو رسوب . فنحتاج أن ننفذ ، أمرا برمجيا لكل حالة من هذه ، ونحن سنجعل أن يطبع لنا المستوى. تابع المثال الآتي.

```
var level = 50                                  50
switch(level){
case 0...50 : println("good")                   "good"
default: println("not good")
}
```

في هذا المثال قمنا بتعريف متغير **level** و أعطيناه القيمة . 50 ثم قمنا بتعريف **switch** ثم قمنا بتحديد المتغير المراد و ضع حالات له، و بعد ذلك أنشأنا { } البلوق. ثم قمنا بكتابة **case** ، بالرجوع إلى المعنى اللغوي للكلمة هذه نجدها تعني حالة ، أي في حالة كانت قيمة المتغير كذا ففعل كذا ، ثم و ضعنا قيمة المتغير فأخبرنا في الحالة الأولى إذا كانت قيمة الـ **level** من 0 إلى 50 نفذ الآتي . و أرجو الانتباه إلى “ : ” ، ثم الولوج في كتابة الكود. و قد تتعجب من وجود كلمة غريبة و هي **default** و هذه الكلمة إجبارية بحيث أنّ و ظيفتها هي في حال لم تتحقق أي حالة من الحالات المكتوبة يتم تنفيذ الكود البرمجي المراد تنفيذه.

حلقات التكرار

ماهي حلقات التكرار! هي عبارة عن تكرار أمر ما أكثر من مرة دون الحاجة إلى كتابة الكود مجددا. فعندما نريد مثلا طباعة الرقم عشرة مئة مرة من غير المنطقي تكرار الكود مئة مرة . لذلك الحل الأنسب حلقات التكرار.

ما هي أنواع حلقات التكرار! نجد بأن حلقات التكرار في لغات البرمجة تنقسم لقسمين. حلقة **While** و حلقة **for** . بحيث أنّ حلقة الـ **for** تتكون من نوعين فيما يتقدم سيتم ذكرهما.

كيف يتم كتابة حلقة while ! نقوم باستدعاء الكلمة **while** ثم نلحقها بالشرط **condition** ثم نضع { } البلوق ، الذي يمكننا من كتابة الكود بداخله. انظر للمثال الآتي.

```
while (condition) {

}
```

مثال على حلقة while ! في هذا المثال سوف نعمل على عمل حلقة تقول في حال أنّ متغير ما أقل من ١٠٠ إذاً قم بزيادة و احد في كل مرة إلى أن يصل إلى الـ ١٠٠ .

```
var n = 0                    0
while  n < 100 {

    n++                      (100 times)
}
```

حلقة while و الـ do ! عرفنا و ظيفة الـ **while** و لكننا لم نفهم ما هي الـ **do** للآن. بالرجوع للمعنى العام إلى الـ **do** نجد بأن المعنى هو " افعل " ، و ذلك بما معناه افعل الكود التالي حينما يتحقق الـ **while** . انظر للمثال التالي.

```
var x = 10

do{
    println("mofeed")
    x--
}while x > 10
```

```
10


"mofeed"
10
```

في المثال الذي في الأعلى استدعينا الـ do ثم جلبنا الـ { } لنضع الكود الذي نود فعله ، و هنا قمنا بإرسال أمر طباعة الـ x في كل مرة يتم تنفيذ الأمر فيها ، يتم إنقاص رقم ١ من القيمة . و بعد ذلك استدعينا الـ while و أعطيناه الشرط في حال x أكبر من صفر ، عُد إلى الـ do و نفذ الأمر مجددا.

حلقات الـ for و الـ for in ! حلقات التكرار هذه تعمل على تنشيط المصفوفات أو تكرار أمر معين عدد من المرات سواء كانت معلومة أو غير معلومة أي لا نهائية .

كيف يتم تكرار أمر معين عدد من المرات! في حال كنا معنين بطباعة أمر معين عدد من المرات نحتاج الـ for . لذلك سوف نقوم بتطبيق أمر معين يمكننا من طباعة كلمة مرحبا عدد معين من المرات. انظر كيف تُبنى هذه العملية.

```
for(beging;condition;Increment){

}
```

في المثال السابق طريقة بناء حلقة التكرار حيث beging تُشير إلى الاستهلال و فيه يتم تحديد المتغيرات . و الـ condition و يقصد بها الشرط و هنا نضع الشرط المراد تنفيذه. و الـ Increment هو معمل الإزدياد أو النقصان . كما في داخل البلوق نضع الأمر . انظر للمثال الآتي . و الذي سوف نطلب من الحاسب تكرار كلمة mofeed ١٠ مرات.

```
for(var i = 0 ; i < 10 ; i++ ){

    println("mofeed")
}
```

```
10

(10 times)
```

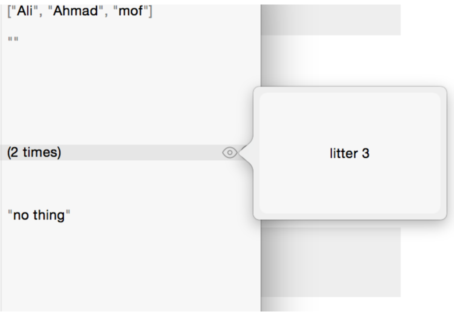
أوامر الإنتقال

مـقدمـة! سـوف نـتعلم في هـذا المـقال الـبسيط ، ثـلاث كـلمات جـداً مـهمة . الأولى **fallthrough** و هـذه الـكلمة تُـعتبر خـاصـة بـقاعـدة **switch** التي تـعلمناهـا سـابـقا ، وسنرى مثالا على ذلك . الكلمة الثانية و هي **continue** و تعني أكمل و سنرى لها مثال أيضاً. الكلمة الثالثة و هي **break** و تعني الكسر . كما نستفيد من هذه الكلمات عدم سير البرنامج وفق نمط التسلسل بحيث نستطيع أن نكسر التسلسل من خلال هذه الكلمات.

الـ **fallthrough** ! هذه الكلمة كما ذكرنا سابقا كلمة خاصة بـ **switch** و التي تكمن وظيفتها في حـال كان لدينا أكثر من فعل لحالة و احدة تقوم بفعله. بحيث الآلية في **switch** تقوم على إذا تحققت حالة واحدة فإنه يعمل عملية كسر لباقي الحالات فلا يقوم بعملية التحقق . لذلك هذه الكلمة جاءت بحل بحيث تعطي الحاسب معلومة بأن يُنفذ لك أكثر من أمر لحالة واحدة. انظر للمثال التالي.

```
var Names = ["Ali" , "Ahmad" , "mof"]
var rs = ""
for name in Names {
    switch count(name)
    {
    case 3 : println("litter 3")
    case 3 : println("mofeed")
    default : println("no thing")

    }
}
```



انـظر لـلمثال الـسابـق اسـتخدمـنا أكـثر مـن درس الـحلقات و الـحالات . و الـدالـة **countElement** دالة لحساب عدد حروف كل قيمة من المصفوفة. و لو تلاحظ طلبنا في حال كانت عدد الحروف ثلاثة نفذ لي أمرين. و لو تلاحظ على يمين الشاشة لم يتم تنفيذ إلا أمر واحد فقط . و هو طباعة النص " **3 leter**" ولم يتم تنفيذ الأمر الذي يتم فيه طباعة "**mofeed**" لذلك تابع الاضافة التالية في المثال الآتي لتفهم لماذا الـ **fallthrough** مهمة

```
case 3 : println("litter 3")
```

litter 3

```
fallthrough
case 3 : println("mofeed")
```

mofeed

لاحظ في المثال السابق لقد قمنا بطباعة الـ **fallthrough** بعد الأمر المراد تنفيذه مباشرة ، لتنتقل للأمر الآخر لتنفيذه.

أكسر و أكمل! لعلنا سنخوض في هذا المثال الكلمتان **continue , break** في مثال واحد ، لكون الكلمتان ترتبطان كثيرا ببعضهما . سنقوم بتطبق هذا الكلام على حلقة **for** بداخلها حلقة **for** أخرى . انظر للمثال الآتي.
في المثال السابق قمنا باستدعاء الدالة الـ **countElements** و التي تعمل على عدْ عدد حروف كل قيمة داخل المصفوفة . بحيث وضعنا شرط في حال عدد الحروف أكبر من ثلاثة **أكمل** الكود داخل ذات الـ { } البلوق و قد قام بتنفيذ الحلقة الثانية بعد ذلك. و يتضح هذا على يمين الشاشة في الـ كانسل آوت بت ، حيث ظهر الاسم Ali . انظر للمثال القادم والذي سيكون عكس هذا المثال تماما فسنضع بدل الـ **continue** في هذه المرة **break** . ستلاحظ أنّهُ لا يوجد أي ناتج في يمين الشاشة و السبب راجع إلى **break** بحيث عندما تحقق الشرط قام بكسر الحلقة الثانية و انتهى مفعولها.

حلات تنشيط المصفوفات ! يتم استخدام حلقات الـ **for in** في مثل هذه الحالة . انظر للمثال الآتي الذي سوف نقوم بتنشط مصفوفة مع حلقة التكرار لعرض ما هو داخل المصفوفة.

```
var arr = ["m" , "a"]

for arr2 in arr
{
    println("\(arr2)")
}
```

```
["m", "a"]

(2 times)
```

```
m
a
```

لا حظ في المثال التالي قمنا بتعريف مصفوفة ثم استدعينا حلقة **for** و أعطيناها اسم **name** ثم اضفنا **in** و التي تشير الى أنّ خذ البيانات التي في المصفوفة **Names** و أضفها إلى **name** . ثم فتحنا البلوق { } ، و طبعنا بداخله الأمر **println** و قلنا له اطبع لنا **name** . بهذا النمط طبقنا حلقات الـ **for** لتنشيط المصفوفات.

```
var Names = ["ali" , "Ahmad"]

for name in Names{

    println(name)
}
```

```
["ali", "Ahmad"]

(2 times)
```

```
ali
Ahmad
```

التعامل مع النصوص

ما هي النصوص المقبولة! سؤال ربما محير نوعا ما . فهل يقبل أي نص! نعم لا تتحير بالفعل عندما تضع نصاً فأنك تستطيع استخدام جميع ما هو موجود على لوحة المفاتيح من رموز و أرقام و حروف.

كيف يتم كتابة النص! تتم كتابة النصوص بتعريف متغير بحيث القيمة فيه تساوي علامة التنصيص بداخلها النص ، " هنا يكون النص ".

```
var str = "هنا النص"
```

طريقة اضافة نص لمتغير يحوي نص! مثلا لو كان لديك متغير يحتوي على اسمك وأردت إضافة اسم العائلة لنفس المتغير دون كتابته مجددا . تابع المثال التالي.

```
var str = "mofeed"
str += " hassan"
```

```
"mofeed"
"mofeed hassan"
```

لاحظ في المثال السابق قمنا بتعريف متغير **str** و أعطيناه اسم مفيد بالانجليزية ثم استدعنا اسم المتغير ووضعناه بعده علامة الـ **+** ثم **=** و التي تشير إلى أنّ القيمة الجديدة التي سوف تُكتب خذها و أضفها للمتغير **str** . لو تتذكر فيما سبق قمنا بشرح كل هذه الدلالات في جدول في درس المتغيرات والثوابت يمكنك الرجوع له.

الدوال "Function"

ماهي الدالة! الدالة عبارة عن مجموعة أوامر توجد في مكان محدد و لها وظيفة محددة و لها اسم كذلك محدد . يمكنك استدعاءها متى شئت من خلال مناداتها باسمها دون الحاجة لكتابة الأكواد المجودة داخلها مرة أخرى.

لماذا الدالة مهمة! الدوال مهمة جدا . فهي من اسمها تدل على الوظيفة . أي قيام الجهاز بأداء و ظيفة مثل ، إنشاء دالة و ظيفتها تستقبل عدد معين و ترجع لنا تربيع العدد.

ما هي البارامترات! الـ **parameter** عبارة عن المتغيرات التي تستقبلها الدالة إن كانت الدالة بحاجة إلى متغير أو أكثر.

كيف يتم إنشاء الدالة! يتم إنشاء الدالة من خلال الكلمة **func** و هي اختصار من الكلمة **function**.

أنواع الدوال! هناك أكثر من نوع للدالة . توجد دالة خالية من البرامترات لكنها تُرجع قيمة ، كما توجد دوال لها برامترات لكنها لا تُرجع قيمة ، و توجد أيضا دوال من غير برامترات و لا ترجع أي قيمة . لا تقلق إن لم تفهم سوف نناقش كل واحدة على حذا لتصلك المعلومة.

دالة لا تستقبل برامترات و لا تُرجع قيمة! مثل هذا النوع من الدوال تكون لأداء شي محدد . أكثر من مرة و في مواضع عديدة. انظر للمثال التالي.

```
func NameTheFunction ()->()
{

    println("mofeed")

}
```

في المثال السابق وضعنا كلمة **func** و التي تُشير إلى الدالة ثم بعد ذلك وضعنا اسم الدالة و هو **NameTheFunction** كما وضعنا بعد ذلك **( )** القوسين و هما يشيران بأن الدالة لا تستقبل بارمترات . ثم بعد ذلك و ضعنا " **->** " وهذه تدل على أنّ الدالة هل سترجع قيمة أم لا ووضعنا بعدها الـ **( )** اشارة بأنّ هذه الدالة لا تُرجع أي قيمة ، كما أخيراً وضعنا البلوق **{ }** . وكتبنا داخله أمر الطباعة على الشاشة بحيث يتم طباعة الاسم مفيد في حال إستدعينا الدالة. انظر للمثال القادم في كيفية استدعاء الدالة.

```
NameTheFunction()

func NameTheFunction ()->()
{

    println("mofeed")
```

"mofeed"

كتبنا اسم الدالة ثم ختمنها بـ ( ) . لذلك تلاحظ في المثال الذي يسبق هذا لم نجد كلمة **mofeed** تم طباعتها قبل استدعاء الدالة.

دالة متعددة البرامترات ! في هذا المثال سنقوم بعرض دالة لها برامترات وحيدة أو عديدة لكنها لا تُرجع أي قيمة

```
func darb (n1:Int , n2:Int)->()
{

    println("\(n1 * n2)")

}
```

.

لاحظ في هذا المثال حددنا برامترات اثنين و حددنا نوعهما أن يكونا عددين صحيحين ، كما و ضعنا داخل البلوق أمر طباعة . و قد يُلفت انتباهك لماذا ( ) \ ؟؟ نعم فلهذه فائدة جدا جميلة ، فهي تُعطيك صلاحية استدعاء متغير أو أكثر وتنفيذ عمليات حسابيه عليهما بداخل النصوص . و هنا أمرنا العملية أن تضرب العددين. انظر لعملية الاستدعاء.
انظر كيف تم استدعاء الدالة من خلال اسمها بالاسفل ، كما ظهرت قيمتان من الاجباري ادخالهما ليتم تطبيق الدالة . انظر لتطبيق الدالة الآتي.

```
darb(4,3)

func darb (n1:Int , n2:Int)->()
{

    println("\(n1 * n2)")

}
```

"12"

طبّقنا الدالة و جعلناها تقوم بعملية ضرب اربعة في ثلاثة و كان الناتج ظاهراً على يمين الشاشة و هو الرقم ١٢ .

دالة تستقبل بارامتر و تُعيد عدة قيم! في هذا المثال سوف نقوم بتطبيق دالة ترجع أكثر من قيمة . بحيث سنبني دالة يتم فيها تحويل الطول لعدة قياسات . انظر للمثال التالي.

```
func tall (high_CM:Float)->(Meter:String , Inch:String)
{
    return ("in meter \(high_CM*0.01)" , "in inch \(high_CM*0.39)")
}

tall(160.0)
```

```
(.0 "in meter 1.6", .1 "in inch 62.4")

(.0 "in meter 1.6", .1 "in inch 62.4")
```

في المثال السابق قمنا ببناء عدة قيم يتم ارجاعها كما عرفنا بارامتر وهو الطول ، و بداخل البلوق عرفنا القيم التي نريد ارجاعها بالمعادلة التي لها. ولكن قد تجد كلمة غريبة و هي **return** ، هذه الكلمة تعني أرجع و هي التي تدل على القيم المراد إرجاعها للمستخدم.

سر من أسرار الدوال! قد لاحظت فيما سبق من أنّ الدالة عندما نستدعيها و نكتب البرامترات لا نرى سوى رقم ، بحيث لو رجعنا لها بعد مدة قد لانعلم ماذا يمثل هذا البارامتر لذلك سوف نكشف الستار عن هذا السر في المثال الآتي.

```
func tall (#high_CM:Float)->(Meter:String , Inch:String)
{
    return ("in meter \(high_CM*0.01)" , "in inch \(high_CM*0.39)")
}

tall(high_CM: 160.0)
```

```
func tall (high_CM:Float)->(Meter:String , Inch:String)
{
    return ("in meter \(high_CM*0.01)" , "in inch \(high_CM*0.39)")
}

tall(160.0)
```

لاحظ الفرق في استدعاء الدالة التي على اليمين و التي على اليسار ، دقق في البارامتر الوحيد للدالة على اليمين يسبقه # ، و هذا هو السر فهذه العلامة تُفيدك في جعل اسم البارامتر متواجد لديك أثناء استدعاء الدالة مما يُسهل عليك عملية فهم الكود و تبسيطه.

الكلوجر closures

ما هو الكلوجر! عبارة عن دالة و لكن لا يشترط أن يكون لها اسم ، و نستطيع استخدام هذا النمط في حال كنا معنيين باستخدام الدالة لمرة واحدة فقط أو في حال كنا معنيين في عمل عمل على سبيل المثال إعادة ترتيب لمصفوفة أرقام بشكل تصاعدي.

كيف يتم كتابة الكلوجر! لابد أن نقوم بحجز مساحة للكلوجرز و ذلك من خلال إنشاء متغير يحمل قيمة الكلوجرز.

هل الكلوجرز مثل الدوال! تماما نفس الدوال و نفس الأنواع كذلك ، فهنالك كلوجر يُعيد قيمة و هنالك كلوجر لا يستقبل بارامتر.

الصيغة العامة للكلوجرز! انظر للمثال التالي.

```
var clus = {(parameters) -> return type in statements}
```

في المثال السابق قمنا بإنشاء متغير ثم فتحنا بلوق **{ }** ثم قوسين و فيهما يتم إدخال البارامترات ثم **->** في حال أنّ الكلوجرز يعيد قيمة نضع مكان **type** النوع و بعد ذلك كلمة **in** الزامية ثم نستبدل مكان **statement** بالكود البرمجي الخاص بك لتنفيذه.

```
var clus = {() -> () in print("this is clousor")}
```

في المثال السابق قمنا بتعريف متغير من خلاله نقوم بتخزين الكلوجر فيه . بحيث أنّ هذا الكلوجر لا يحمل أي بارامتر و كذلك لا يقوم بإرجاع أي قيمة.

كلوجر يُرجع قيمة و يستقبل بارامترا! سوف نقوم بتطبيق هذا على مصفوفة بحيث سنعمل على اضافة قيم عشوائية التصنيف ونعيد تصنيفها مجددا حسب الترتيب التصاعدي.

```
var arr = [1, 7 , 8 ,5 , 3]          [1, 7, 8, 5, 3]

arr = sorted(arr , {a,b in return a<b})   (10 times)

arr                                  [1, 3, 5, 7, 8]
```

قمنا بتعريف مصفوفة عشوائية ثم أضفناها إلى متغير لديه دالة الترتيب **stored** ثم أضفنا الكلوجر الذي وضيفته ترتيب المصفوفة من الكبير للصغير ، كما الكلوجر يحتوي على بارامترين هما **a , b** كما توجد كلمة in و التي تُشير إلى أنّك الآن داخل البلوق واكتب ما تريد ، ثم كتبنا أعد **return** الـ **a** اصغر من الـ **b** .

الـ enumeration

الـ enumeration هو! عبارة عن نوع من القيم تكون لها علاقة مع بعضها في شكل مجموعة . كما يُستخدم أيضاً في عملية الترتيب للمعلومات.

كيف يتم كتابته! لإنشاء **enumeration** جديد نقوم بكتابة **enum** ثم نلحقه الاسم ، وبعد ذلك نفح البلوق كما في الـ **enum** سنستخدم **case** لكل تصنيف.

مثال على الـ enum ! سوف نقوم بإنشاء enum يحتوي على صنفين الرجل و المرأة ، و لكل منهما متغيرات يتم ادخالها مثل العمر و سنة الميلاد. انظر للمثال التالي.

انظر للمثال الموجود على يسار الصفحة و هنا تم إنشاء **enum** يحمل الاسم **Gender** كما يحتوي على قيمتين داخله **Male** , **Fmale** ، كما السطر الأخير يشير لكيفية استدعاء الـ **enum** .

```
enum Gender {

    case Male
    case Fmale

}

    Gender.Fmale
```

الـ enum مع قاعدة switch ! في المثال الآتي سوف نقوم بعمل **enum** للايام ، كما أنّ وظيفة **switch** سوف تكون في حال كان اليوم هو الفلاني قم بطباعة الاسم لذلك اليوم.
في المثال السابق قمنا بعمل اختصار لـ **case** بحيث جعلناها كلها في سطر واحد . ثم قمنا بتعريف متغير من نوع **enum** و بعد ذلك استدعينا احدى القيم. ثم قمنا بعمل الـ switch ثم و ضعنا الحالات المكنة مع تنفيذ أمر لكل واحد منهم.

```
enum Days {
    case Sabt , Ahad , Ethnin , Tholatha , Arbeaa , Kamees , Jumaah

}

var Example:Days = Days.Arbeaa                                        (Enum Value)

switch(Example){

    case Days.Sabt : println("Sabt")
    case Days.Ahad : println("Ahad")
    case Days.Ethnin : println("Ethnin")
    case Days.Tholatha : println("Tholatha")
    case Days.Arbeaa : println("Arbeaa")                              "Arbeaa"
    case Days.Kamees : println("Kamees")
    case Days.Jumaah : println("Jumaah")

    default:println("not good")

}
```

متغيرات محلية و غير محلية

المتغيرات المحلية local ! المتغيرات المحلية هي المتغيرات التي تكون في نطاق غير واسع خلال البرنامج ، أي محددة المكان و لا يمكن استدعاءها في مكان آخر. مثلا عندما نقوم بإنشاء دالة و نضع بداخلها متغير ما ، و نريد الوصول له خارج الدالة لإضافة تعديل عليه فلن نستطيع ، لأنه ينطبق عليه مسمى متغير محلي.

```
func Local_Var() {
    var x = 9
}
  x = 1
```

نلاحظ أنّه لا يمكن التعديل على **x** لكون المتغير محلي .

المتغيرات الغير محلية! الـ **global** هي المتغيرات التي تكون في نطاق واسع ، أي من الممكن استخدامها في أي موقع من مواقع الكود البرمجي . و تتم تعريف المتغيرات هنا خارج الدوال أو خارج الفئات "لا تقلق بشأن الفئات الآن" .

```
var x = 9
func Local_Var() {
    x = 7
}
    x = 1
```

نلاحظ عندما قمنا بتعريف المتغير **x** خارج نطاق الدالة أعطانا حرية لتغير قيمة المتغير في أي جهة من الكود.

المباني و الفئات

مقدمة! في البداية يجب فهم ماهي الفئات **class** و ماهي المباني **struct** في لغة البرمجة المدعاه كائنية التوجه. المباني و الفئات عبارة عن قوالب بداخلها تحوي على الخصائص و الأفعال ، و يمكنك اشتقاق كائن **object** منها . فعلى سبيل المثال نأخذ البشر كفئة و يتسلسل منها أحمد و محمد و لكل منهما خصائص المختلفة و الخصائص المتشابهه فعلى سبيل المثال لون البشرة تعتبر صفة تختلف بينهما ، لكن من الصفات الثابته عدد الأعين.

المباني! لعل ما يميز الباني عن الفئات أنها لا تستقبل عمليات الوراثة و التعدد الصوري ، كما تعتمد على مبدأ **ByValue** . كما بالفعل أنها جدا تشبه الفئات من ناحية الفكرة.

كيف يتم كتابة المباني! تتم كتابة المباني باستخدام الكلمة **struct** ثم بعد ذلك نضع اسم المبنى مباشرة و نختمها بوضع البلوق { } . انظر للمثال الآتي.

```
struct Name_Of_Struct {
    //properties
    //methods
}
```

انظر للصيغة العامة لبناء المباني بالأعلى ، انظر للكلمات المجودة بداخل المبنى , **properties & methods** الخصائص و الأفعال "السلوك".

الـ properties ! و هذه الكلمة تعني الخصائص فمثلا لو أنشأنا مبنى باسم **المدرسة** فنحتاج لتعريف الخصائص مثل عدد طلابها أو عدد المعلمين فيها عدد المدراء و الفراشين و اسم كل منهم .

الـ methods ! الأفعال ، وقد مر عليك من قبل هذا الشيء و لكن بمسمى آخر . الـ **methods** عبارة عن الدوال داخل المباني أو الفئات على حدٍ سواء.

المستطيل الملون باللون الرصاصي عبارة عن الخصائص للمبنى **School** ، كما يتضح في الخاصية رقم ثلاثة أنها اختيارة أي من الممكن أن تكون فارغة **nil** ، كما تلاحظ في الاسفل هنالك متغيرين جديدين جعلانهم يكسبون الخصائص من المبنى **School** بحيث يُعتبر في هذه الحالة أنّ كل متغير الآن هو كائن من نوع المبنى **School** بحيث أعطاك هنا دالة جاهزة مع بناء المبنى.

```
struct School {
    var name_school:String!
    var number_phone:Int!
    var number_of_student:Int?
}

var s1 = School(name_school: "mofeed", number_phone: 23, number_of_student: nil)
var s2 = School(name_school: "mofeed", number_phone: 23, number_of_student: 89)
```

كيف يتم الوصول لقيمة داخل الكائن! من خلال وضع النقطة ثم اسم الخاصية بعد استدعاء اسم الكائن. انظر للمثال التالي.
استدعينا اسم أحد الكائنات المنشأة سابقا ، ثم و ضعنا اسم الخاصية لنحصل على القيمة التي بداخلها.

```
s1.name_school        "mofeed"
```

مبدأ ByVal ! هذا المبدأ يكون في المباني و لا يكون في الفئات انتبه لهذه المعلومة جيدا ، هذا المبدأ ببساطة تتم فيه قيم المتغيرات التي تكسب قيمتها من متغير آخر على نفس القيمة الأساسية لا تتغير حتى بعد تغيير قيمة المتغير الذي كسبنا منه القيمة. انظر للمثال التالي.

```
var x = 1        1
var c = x        1
x = 5            5
c                1
```

لاحظ في المثال السابق قمنا بتعريف متغير **x** و أعطيناه القيمة **1** كما قمنا بتعريف **c** و الذي يساوي **x** و في السطر الثالث أعطينا اضافة للقيمة **x** بحيث أصبحت =**5** و في السطر الأخير طبعا قيمة **c** و لاكنها لازالت تساوي **1** . و هو هذا مبدأ الـ byVal.

القيم المحسوبة

الـ **Computed Property** ! القيم المحسوبة عبارة عن قيمة داخل المباني و الفئات و التي تعتمد على قيمة مسبقة نُدخلها كبرامتر للدالة التي تُعطينا اياها لغة البرمجة تلقائيا. و هما كلمتان **set** , **get** .

```
struct salary {

    var Monthly:Float
    var Yearly:Float
        {
        get{ return Monthly*12}
        set (NewYearSalary) { Monthly =
            NewYearSalary/12  }
    }


}

var s1 = salary(Monthly: 12000)
s1.Yearly
s1.Yearly = 130000
```

```
144000
{Monthly 10833.33}

{Monthly 12000}
144000
{Monthly 10833.33}
```

قمنا بتعريف مبنى **salary** الذي يحوي على خاصيتين هما **Yearly** , Monthly ، لكن لو تلاحظ أنّه الـ **yearly** لها بلوق { } و هو لكتابة المقطعين **get** , **set** ، الـ **get** هنا يُفيدنا في إرجاع قيمة **Monthly** مضروبة في ١٢ و هو عدد الشهور لإستخراج الراتب السنوي ، أما مقطع الـ **set** فيفدنا في أخذ القيمة الجديدة الناتجة عن **yearly** و يقسمها على ١٢ . فالفائدة من **get** احداث تغير و زرع قيمة في الـ **yearly** اما الـ **set** فهو مسؤول عن حال استدعاءك للـ **yearly** خلال الكود و إعطاءه قيمة معينة بإحداث أمر ما . لاحظ في المثال السابق أنشأنا كائن اسمه **s1** ، من نوع مبنى الراتب و أعطيناه القيم الخاصة. ولاحظ أننا لن نجد الخاصية **yearly** لأنها قيمة محسوبة تعتمد على الـ **Monthly** ، لذلك عندما استدعينا الـ yearly الخاصة بالكائن s1 وجدنا قيمتها ١٤٤٠٠٠ ، و عندما اردنا التعديل على الـ **yearly** نجد على يمين الشاشة تم تغير قيمة الـ **Monthly** تلقائيا . ففي حال التعديل اشتغل المقطع **set** .

الخصائص الكسولة

الـخـصـائـص الـكـسـولـة ! و الـتي يُـطلق عـليها **lazy** و الـتي بـشـأنـها تـعمل عـلى تـخـفيف الحمل على الذاكرة. فهنالك أكواد مثل الاتصال بالسيرفر على الشبكة تكون معقدة التركيب وتستهلك حجم طائل من الذاكرة و في الأحيان لا نحتاج لها ، لذا جاءت فكرة الخصائص الكسولة المعنية بحجز مساحة في الذاكرة لخاصية عن الحاجة لها فقط. مما يُعطينا حرية لـجعل الـخـاصـيـة أن تـكـون فـارغـه أي **nil** ، و تـذكـر مـن الـضـروري إعـطـاء الـقـيمة الأولـية للخاصية الكسولة و إلا فالمترجم سوف يُخبرك عن خطأ ما .
لاحظ كيف تم إنشاء الخاصية الكسولة ، في مبنى المدرسة ، ثم استدعاء الكلمة **lazy** ثم عرّفنا الخاصية.

```
struct school {

        lazy var phone_number = 050000

}

school(phone_number: Int? )
```

الخصائص المُراقِبَة

ما هي الخصائص المراقبة property Observe ! قد تكون لاحظت من ذي قبل برنامج يطلب منك إدخال عمرك ، وحينما وضعت عمر غير حقيقي و كان أصغر منك لم يقبله البرنامج ، هذا ما يسمى بـ الخصائص المراقبة.

ماهي الـ didSet , willSet ! هما عبارة عن منظومة المراقبة للخصائص داخل الكود البرمجي ، و الـ **willSet** و هي الخاصة بالقيمة الجديدة التي سيقوم المستخدم بإدخالها و الـ **didSet** و هي الخاصة بالقيمة القديمة أو السابقة.

```
struct MyYears {

    var age:Int = 1
        {
            willSet{" newValue \(newValue)"}            (2 times)

            didSet{ " oldValue \(oldValue)"             (2 times)

                if age>100 || age<0
                {
                    age = oldValue                      {age 9}
                }

            }
        }

}

var s = MyYears(age: 9)                                 {age 9}
s.age = -8                                              {age 9}
s.age = 90                                              {age 90}
```

في المثال السابق قمنا بإنشاء مبنى العمر ثم قمنا بعريف الخاصية **age** ثم وضعنا لها بلوق { } لنضع مقطعي الـ **didSet , willSet** ثم قمنا بتعريفهما دخل لبلوق مع بلوق خاص لكليهما . لا تقلق بشأن الكلمتين **oldValue, newValue** فالكلمة الـ **newValue** لا يمكن استخداها إلّا في الـ **willSet** وتعني القيمة الجديدة ، و بالنسبة للـ **oldValue** فلا يمكن استخدامها إلا في الـ **didSet** . ثم بعد ذلك أنشأنا الشروط التي تُخبرنا إذا كان العمر أقل من صفر أو أكبر من ١٢٠ فاطبع القيمة المُدخله ، وكما تلاحظ في آخر سطر من الكود وضعنا القيمة سالبة و هي أقل من الصفر لذلك تم استخدام القيمة السابقة .

الخصائص الخاصة بالمباني

ماهي الخصائص الخاصة! هي الخصائص التي لا يستطيع أي كائن أن يكسبها ، و التي لا تكون على هيئة برامتر. أي أنّها خاصة بالمبنى فقط.

كيف يتم كتابتها! تتم كتابة الخاصية الخاصة باسباقها بالكلمة **static** .

```
struct AddFiveToAge {

     var TheAge:Int = 1
    static var add5:Int = 5
    var MyAge:Int{

        get{ return TheAge + AddFiveToAge.add5}          95
    }

}

var s = AddFiveToAge(TheAge: 90)                         {TheAge 90}
s.MyAge                                                  95
```

في المثال السابق قمنا بتعريف مبنى يحمل الاسم **AddFiveToAge** و الوظيفة من هذا المبنى هي أن نقوم بإضافة خمس سنوات على العمر الذي نُدخله. بعد ذلك عرفنا ثابت من نوع **static** و الذي يجعله خاصاً في المبنى و ليس الكائن المتفرع من المبنى ، أي أنه لا نستطيع الوصول له عبر الكائن بل عن طريق المبنى فقط. بعد ذلك قمنا بتعريف متغير **TheAge** و بعد ذلك عرفنا المتغير **MyAge** و أعطيناه قيمة محسوبة "مقطع **get**" و أخبرناه بإرجاع **TheAge + AddFiveToAge.add5** ، لا حظ كيف تم استدعاء الخصائص الخاصة من خلال مناداة اسم المبنى ثم نقطة ثم اسم الخاصية.

لاحظ المثال السابق أنشأنا كائن من المبنى و أعطيناه قيمة العمر 90 . و في السطر الأخير استدعينا الخاصية المحسوبة MyAge لنرى على يمين الشاشة تمت طباعة العمر 95 . و نلاحظ هنا أنه عندما نقوم باضافة ٥ سنوات لـ 90 تصبح 95 .

الدوال و المباني

قد سبق شرح ما هي الدوال في الدروس السابقة و عرفنا الفائدة من وراء هذه الدوال . و إن صح الـتعبير فهي تسمى methods مـا دامـت في إطـار المـباني أو الـفئات . انـظر للمثال الآتي .

```swift
struct info {

    var age:Int
    var name:String

    func getInfo()->String{

        return "my age : \(age) , my name : \(name)"
    }


}

var s = info(age: 20, name: "mofeed")

    s.getInfo()
```

```
"my age : 20 , my name : mofeed"

{age 20, name "mofeed"}

"my age : 20 , my name : mofeed"
```

في المثال السابق قمنا بتعريف مبنى باسم **info** بداخله خاصيتان و هما الاسم و العمر ، ثمّ قمنا بإنشاء دالة **method** و التي تسمى **getInfo** بحيث أنّ الدالة تُرجع قيمة نصية ، والقيمة النصية تساوي العمر بالاضافة للاسم. ثم قمنا بإنشاء كائن اسميناه s من نوعية info و من ثـم قمنا بـتطبيق الـدالة الـجاهـزة و الـتي تـتظمن الـعمر و الاسـم ، وفي الـسطر الاخير استدعينا الدالة getInfo .

المباني و mutating

```
struct info {

    var age:Int
    var name:String

    func getInfo()->String{

        return "my age : \(age) , my name : \(name)"
    }

   mutating func editAge(num:Int){

        age = num

    }


}

var s = info(age: 20, name: "mofeed")

    s.editAge(50)
    s.getInfo()
```

"my age : 50 , my name : mofeed"

{age 50, name "mofeed"}

{age 20, name "mofeed"}

{age 50, name "mofeed"}
"my age : 50 , my name : mofeed"

قمنا في المثال السابق قمنا باضافة دالة جديدة و هي editAge ووظيفة هذه الدالة التعديل على العمر الخاص بالكائن ، وبما أنّ المباني تتبع مبدأ byVal لذلك لن يقبل الكومبايلر age = num لانه يعتبر 50 = 20 و هذا غير منطقي لذلك ترى الكلمة التي تسبق الدالة و هي mutating تحل لنا الموضوع هذا فتجعل الأمر جدا مقبول.

سر كلمة self

```
struct info {

    var age:Int
    var name:String

    func getInfo()->String{

        return "my age : \(age) , my name : \(name)"
    }

   mutating func editAge(age:Int){

       self.age = age

    }


}

var s = info(age: 20, name: "mofeed")

    s.editAge(50)
    s.getInfo()
```

```
"my age : 50 , my name : mofeed"

{age 50, name "mofeed"}

{age 20, name "mofeed"}

{age 50, name "mofeed"}
"my age : 50 , my name : mofeed"
```

لقد قمنا بإضافة بعد التعديلات على الكود في الدرس السابق ، فلو تلاحظ المستطيل الاحمر أننا قمنا بإضافة **self** للخاصية age و كذلك قمنا بتغير num الى age لكي نمنع أنفسنا من ابتكار الكثير من الأسماء ذات المعنى الواحد في مثل هذه الحالات . إذا ماذا تعني كلمة **self** قبل الـ age ببساطة فهذه الكلمة تعني أنّ هذا المتغير خاص بالمؤشر أو خاص بالمبنى و ليس خاص بالدالة ، مما يتيح لك أن تقوم بإضافة نفس الاسم كبرامتر في الدالة و كخاصية من خواص المبنى .

الـ Subscript

ماهو الـ Subscript ! إن صح التعبير فهو عبارة عن اختصار لإستدعاء بعض القيم داخل المباني و الفئات من خلال الحاق اسم الكائن بـ [ ] و وضع داخله رقم الاندكس. كما من الممكن استخدام أكثر من Subscript و لكن بشرط إختلاف التواقيع أي البرامترات المستقبلة . انظر للمثال التالي.

```
struct info {

    var age:Int
    var name:String

    func getInfo()->String{

        return "my age : \(age) , my name : \(name)"
    }

    subscript(i:Int)->String
    {
        if i == 0 {return name}
        return " "
    }

}

var s = info(age: 20, name: "mofeed")

    s[0]
```

```
"mofeed"

{age 20, name "mofeed"}

"mofeed"
```

دوال البناء

مـاهـي دوال الـبنـاء Initialization ! بـبساطـة و مـن غير تـعقيد هي عـبارة عـن دالـة تحمل اسم المبنى أو الفئة ، بحيث يمكن استدعاءها مباشرة عندما نشتق كائن من المبنى أو الفئة. و في الغالب نستخدمها لإعطاء قيم أولية للخصائص. و كذلك فهذي الدوال لا تقوم بإرجاع أي قيمة .

كيف يتم كتابة هذه الدالة! تتم كتابة الدالة من خلال الكلمة المحجوزة **init** .

```
struct Celsius {
    var temperatureInCelsius: Double

    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) /
            1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }


}

    let t = Celsius(fromFahrenheit: Double)                {temperatu
        M  Celsius (fromFahrenheit: Double)
        M  Celsius (fromKelvin: Double)
```

في المثال السابق قمنا بتعريف مبنى و بداخله خاصية الحرارة بالسيليسوز ، و بعد ذالك قمنا ببناء دلتين بناء و كل منهما يحمل بارامتر واحد فقط ، و استناداً لمعادلة معينة طلبنا أن تكون القيمة لكلا الدالتين . ثم انتقل للسطر الاخير الذي فيه قمنا باشتقاق كائن من المبنى Celesius و قمنا بتطبيق دوال البناء الموجودة فيه و هما اثنتان.

ماهو الـ class

ما هو الكلاس! إن صح التعبير فهو عبارة عن صندوق أو حاوية بداخله مجموعه من الصفات و الدوال متعددة المهام أي أنه لا يشترط كل دالة تعمل ذات العمل ، بل هي مختلفة ، كما لكل كلاس اسم تقوم بتحديده أنت ، و من خلاله تستطيع مناداة الخصائص و الدوال ، كما يُتيح لك إشتقاق كائنات منه.

ما الفرق بين المباني و الكلاس! يكمن الفرق في شمولية الكلاس (الفئة) على عمليات التعدد الصوري و الوراثة و كثير من هذه الأمور التي لا يحويها المبنى ، كما أنّ المباني لطيفة في الاستخدامات البسيطة ، و أهمية أفضلية الكلاس عليها ، إلا أنّ للمباني بعض المميزات التي تفتقدها الكلاسات.

كيف تتم كتابة الكلاس! تتم كتابة الكلاس من خلال الكلمة المحجوزة class و من ثم نُلحقها باسم الكلاس و هوه اختياري حسبما تريده و من ثم { } . انظر للمثال التالي.

```
class name {
    properties and methods
}
```

كما ترى في المثال العلوي ، الصيغة العامة للفئات ، كما ترى بداخل البلوق { } كلمتان تم ذكرهما في المباني و هما الخصائص و الدوال.

القيم المحسوبة

الـ **Computed Property** ! القيم المحسوبة عبارة عن قيمة داخل المباني و الفئات و التي تعتمد على قيمة مسبقة نُدخلها كبرامتر للدالة التي تُعطينا اياها لغة البرمجة تلقائيا.
و هما كلمتان **set** , **get** . تماما كما في المباني.

```
class AVageS {

    var maxStudent:Int! , minStudent:Int!

    var Avrage:Int{

        get{return (maxStudent + minStudent) / 2}
    }


}

let s = AVageS()                                            {nil nil}
        M  AVageS ()
```

كما تلاحظ لقد قمنا بتطبيق القيم المحسوبة و عندما قمنا باشتقاق كائن لم تظهر لي دالة جاهزة كما في المباني ! و هذا يأخذنا لموضوع دوال البناء و هو الدرس القادم.

دوال البناء في الكلاس

ماهي دوال البناء Initialization ! ببساطة و من غير تعقيد هي عبارة عن دالة تحمل اسم المبنى أو الفئة ، بحيث يمكن استدعاءها مباشرة عندما نشتق كائن من المبنى أو الفئة. و في الغالب نستخدمها لإعطاء قيم أولية للخصائص. و كذلك فهذي الدوال لا تقوم بإرجاع أي قيمة .

هل الكلاس يعطي دالة بناء جاهزة! الكلاس لا يُعطي أبداً دوال بناء جاهزه بل تحتاج أنت أن تقوم ببنائها يدويا. انظر للمثال القادم.

```
class AVageS {

    var maxStudent:Int! , minStudent:Int!

    var Avrage:Int{

        get{return (maxStudent + minStudent) / 2}
    }

    init(maxStudent:Int , minStudent:Int){

        self.maxStudent = maxStudent
        self.minStudent = minStudent


    }

}

let s = AVageS(maxStudent: Int, minStudent: Int)
    M AVageS (maxStudent: Int, minStudent: Int)
```

في المثال السابق قمنا بإنشاء كلاس و ظيفته يُخرج لنا المتوسط الحسابي لعدد طلاب مدرسة ما ، و لكن لولا وجود دالة البناء لما كان باستطاعتنا أن نحصل على الدالة التي باللون الأزرق ، و أيضا لو تلاحظ بداخل الدالة كلمة سلف ارجع لدرس **سر كلمة self** فهي بذات المعنى.

```
let s = AVageS(maxStudent: 120, minStudent: 95)

    s.Avrage
```

الدوال الخاصة بالفئة

هي الدوال التي نستدعيها من خلال اسم الفئة و ليس من خلال الكائن المشتق من الفئة.

كيف تتم كتابتها! تتم كتابة الدوال الخاصة في الفئة من خلال الكلمة **class** لو تقرانها في المباني فقد كانت **static** .

```
class AVageS {

    var maxStudent:Int! , minStudent:Int!

    var Avrage:Int{

        get{return (maxStudent + minStudent) / 2}                (2 times)
    }

    init(maxStudent:Int , minStudent:Int){

        self.maxStudent = maxStudent
        self.minStudent = minStudent

    }

    class func WhereAkbar(s1:AVageS , s2:AVageS)->Bool{

        return s1.Avrage>s2.Avrage                                true
    }

}

let ss1 = AVageS(maxStudent: 12, minStudent: 23)                  {maxStudent 12 minStudent 23}
let ss2 = AVageS(maxStudent: 67, minStudent: 2)                   {maxStudent 67 minStudent 2}

AVageS.WhereAkbar(ss2, s2: ss1)                                   true
```

في المثال السابق لقد قمنا بإنشاء الدالة WhereAkbar و التي وظيفتها تعمل مقارنه بين كائنين من نفس الفئة و ترجع قيمة true اذا كان المتوسط الحسابي الاول أكبر من الثاني. انظر للسطر الأخير تطبيق لهذا الكلام.

الوراثة Inheritance

ماهي الوراثة! لو قُلنا بأن شركة أبل على سبيل المثال لديها أبناء هما ماك بوك وآي ماك بحيث هؤلاء الأبناء يرثون الصفات من الأب أبل مع تفرد كل إبن بخصائص خاصه فيه ، هذا ببساطة مبدأ الوراثة في البرمجة كائنية التوجه.

```
class phones {

    var name:String!
    var model:String!

    init(name:String , model:String){

        self.name = name
        self.model = model

    }

}

class IPhone4s:phones {

    var has_camera:Bool!

    init(name:String , model:String , has_camera:Bool){

        super.init(name: name, model: model)

        self.model = model
        self.name = name
        self.has_camera = has_camera

        println("model : \(model) - name : \(name) - has camera : \
            (has_camera)")

    }

}

let pho = IPhone4s(name: "4s", model: "2015", has_camera: true)
```

في المثال السابق قمنا بإنشاء كلاس اسمه phones بداخله خصائص الاسم و الموديل مع الدالة الخاصة بالبناء له ، ثما قمنا بعمل كلاس اسمه Iphone4s و ألحقنها بـ : ثم اسم الكلاس السابق . وبهذه الطريقة جعلنا الكلاس Iphone4s يرث من الكلاس phones بالتالي أصبحت الصفات مورثة لديه ، كما قمنا الآن باستحداث خاصية جديدة و هي has_camera ، و من ثم قمنا بإنشاء دالة بناء تحوي على جميع الخواص الموثه من الأب و الخاصة بالكلاس الجديد الذي يرث ، و لو تلاحظ و جود كلمة **super.init** ،

ووجدها يعني أننا سوف نقوم باستخدام الصفات الموجودة في الكلاس الأب من خلال كلمة self ، و في نهاية الدالة طلبنا بطباعة البيانات التي سوف نحصل عليها. لا حظ السطر الأخير لقد قمنا باشتقاق كائن من الكلاس الإبن.

مبدأ التحميل الزائد

ماهو مبدأ Overloading ! هذا المبدأ جدا سهل الفهم و ليس معقد أبداً ، كل ما يعنيه هذا المبدأ هو ، أن يكون لديك مجموعة من الدوال تحمل ذات الإسم و لكن بشرط واحد أنّ التوقيع يكون مختلف ، و نقصد بالتوقيع هو إما عدد البرامترات أو إختلاف أسماء البرامترات في الدالة.

```
class phones {

    func pr(name:String){

        println(name)
    }

    func pr(age:Int){

        println(age)
    }

}
```

مبدأ الـ Overriding

ماهو الـ Overriding ! ببساطة لو كانت لديك دالة في كلاس الأب و أردت التعديل عليها في كلاس الإبن ، من حيث الوظيفة أي أن التوقيع لا يختلف فنحتاج هذه الكلمة. انظر للمثال التالي.

```
class phones {

    func pr(name:String){

        println(name)
    }

}

class mm:phones {

    override func pr(name:String){

        println("name : \(name)")

    }

}
```

الكلمة final

بما أننا عرفنا مبدأ **overriding** و فهمناه ، نأتي للكلمة **final** و التي تعمل على حماية الدالة من **overriding** أي أننا نقوم بوضع الكلمة **final** قبل الوظيفة في الكلاس الأب ، و بالتالي عندما نأتي لكلاس الإبن لنقوم بإجراء التعديل عليها فإننا لا نستطيع و السبب وجود كلمة **final**.

```
class phones {

   final func pr(name:String){

        println(name)
    }

}

class mm:phones {

    override func pr(name:String){
                            Instance method o
        println("name : \(name)")

    }

}
```

التعدد الصوري

ماهو الـ polymorphism ! التعدد الصوري إن صح التعبير هو مبدأ من مبادئ الوراثة يكون فيه نوع المؤشر مختلف عن نوع الكائن ، بحيث أنّ لدينا الكلاس الأب يورث للأبناء ذات الصفات بالإضافة لصفات خاصة لكل إبن ، و لكننا عندما نقوم بتطبيق هذا المبدأ سوف نحصل على الصفات المشتركة بين الإبن و الأب فقط ، كما يمكن عمل override للدوال.

```
class a {
    var m:Int!
    init(m:Int){
        self.m = m
    }
    func up(m:Int){
        self.m = m
        println(m + 2)
    }

}
class B:a {
    override func up(m:Int){
        self.m = m
        println(m + 3)
    }
}
class C:a {
    override func up(m:Int){
        self.m = m                                  {{m 9}}
        println(m + 4)                              "13"
    }

}
let bb:a = C(m: 9)                                  {{m 9}}
    bb.up(9)                                        {{m 9}}
```

دالة الهدم

ماهي دالة الهدم! ببساطة شديدة و هي دالة تحرير الذاكرة من جميع العمليات الخاصة بالكائن ، عندما يساوي **nill** .

كيف يتم الافصاح عنها! يتم الافصاح عنها بالكلمة **deinit** .

شروط! نعم توجد شروط لدالة الهدم فهي لا تستقبل برامترات و لا ترجع أي قيمة إبداً كما أنّه كل كلاس يحتوي على دالة هدم واحدة فقط ليس أكثر ، و أيضا لا نستطيع عمل **override** لهذه الدالة.

```
class dr {

    var num1:Int!
    var num2:Int!

    init(n1:Int , n2:Int){

        self.num1 = n1
        self.num2 = n2

        println("\(num2 * num1)")
    }

    deinit{

        println("تم تحرير الذاكرة")
    }

}

var s:dr? = dr(n1: 1, n2: 1)

s = nil
```

```
"1"
"تم تحرير الذاكرة"
{num1 1 num2 1}
nil
```

```
class dr {

    var num1:Int!
    var num2:Int!

    init(n1:Int , n2:Int){

        self.num1 = n1
        self.num2 = n2

        println("\(num2 * num1)")
    }

    deinit{

        println("تم تحرير الذاكرة")
    }

}

var s:dr? = dr(n1: 1, n2: 1)
```

```
"1"
{num1 1 num2 1}
```

في المثال الذي على اليسار تم تنفيذ أنّ المتغير يساوي nil بالتالي على تمت طباعة كلمة تم تحرير الذاكرة ، بينما في المثال الذي على اليمين لم يتم طباعتها و السبب أن الكائن لا زال غير فارغ أي أنه ذات قيمة.

علامتي ؟ و !

ما الفرق! هذه العلامتان نضعهمها بعد تحديد نوع المتغير ، و الفرق بينهما إنّه إذا تم وضع علامة ؟ بعد نوع المتغير ذلك يعني أنّ المتغير قد يكون يساوي nil اي لاشيء وقد يكون يحمل قيمة بدخله ، و على العكس تماما فعلامة ! تعني حتميا توجد قيمة بالداخل.

كلمات weak , strong , unowned

مقدمة! هذه الكلمات الثلاث لها علاقة بتحرير الذاكرة من الخصائص التي في الذاكرة لأحد المؤشرات.

كلمة strong ! و هي الكلمة التي تكون موجودة بشكل تلقائي من غير كتابته للخواص داخل الكلاس ، و هذه الكلمة تعني أنه لايتم تحرير الخاصية من الذاكرة .

كلمة weak ! و هذه الكلمة تعني الضعف و على عكس الكلمة السابقة تماما فهي تقوم بتحرير الخاصية من الذاكرة بشرط أن تكون اختيارية أي أنه يوجد قبلها ? .

كلمة unowned ! مثل weak تماما لكنه لا يشترط لها علامة ? .

```
weak var m:dr?

unowned var c:dr
```

الكلمتان anyobject , any

الكلمة any-object ! هذه الكلمة تتيح لنا إنشاء مصفوفة تحوي على مؤشرات من نفس النوع و لا تشترط لنا أن نعرف نوعها لكن لابد أن تكون من نفس النوع .

```
class dr {

    var num1:Int!
    var num2:Int!

    init(n1:Int , n2:Int){

        self.num1 = n1
        self.num2 = n2

        println("\(num2 * num1)")
    }

    deinit{

        println("تم تحرير الذاكرة")
    }


}

var x:dr = dr(n1: 0, n2: 6)
var xx:dr = dr(n1: 9, n2: 6)

var arr:[AnyObject] = [x , xx]
```

الكلمة any ! هذه الكلمة تتيح لك استقبال أكثر من نوع في المصفوفة ، نصي أو رقمي أو من كائن مشتق من كلاس.

```
class dr {

    var num1:Int!
    var num2:Int!

    init(n1:Int , n2:Int){

        self.num1 = n1
        self.num2 = n2

        println("\(num2 * num1)")
    }

    deinit{

        println("تم تحرير الذاكرة")
    }


}

var x:dr = dr(n1: 0, n2: 6)
var xx:dr = dr(n1: 9, n2: 6)

var arr:[Any] = [x,9,"mofeed"]
```

ماهو الـ protocol

ماهو البرتكول ! عبارة عن حاوية تحوي على العمليات والخصائص المراد من الكلاس أو المبنى تنفيذها بشكل مؤكد و إلزامي ، بشرط كتابة الدالة وعدم تنفيذها إلا بداخل الكلاس.

كيف يتم كتابة البرتكول! من خلال الكلمة المحجوزة **protocol** و من ثم إلحاقها باسم البرتكول و من ثم بالبلوق { } .

ملاحظة! البرتكول لا يقبل الصفات العادية لكنه يقبل الصفات التي تحمل على مقطع **get** و **set** أي المحسوبة.

```
protocol MyProtocol {
    var MySalary:Int { get }
    func MyName()->String
}
```

في المثال الذي بالأعلى قمنا بإنشاء برتكول و الذي يحمل الاسم **MyProtocol** و الذي بداخله الصفات الإلزامية.

هل من الممكن وضع عمليات اختياريه! نعم نسطيع ذلك بإضافة كلمة **optional** قبل العملية ، و لكن ذلك ليس كافيا ليفهم البرتكول أن لديك عملية اختيارية ، نحتاج إلى إضافة الكلمة **@objc** قبل البرتكول.

```
import Foundation
@objc protocol MyProtocol {
    var MySalary:Int { get }
    func MyName()->String
    optional func myAge()->Int
}
```

كيف يتم تنفيذه على الكلاس! يضاف بعد اسم الكلاس : ثم نلحقه باسم البرتكول.

```
class person:MyProtocol {

}
```

تطبيق البروتكولات ! انظر للمثال التالي.

```
import Foundation
 protocol MyProtocol {
    var MySalary:Int { get }
    func MyName()->String

}
@objc protocol Mypro {
    optional func myAge()->Int

}
class person:MyProtocol , Mypro {
      var MySalary:Int { return 900 }              900
      func MyName()->String{
         return "mohammad"                         "mohammad"
    }
    func myAge()->Int{
         return 90                                 90
    }
}
let p = person()                                   __lldb_expr_1485.person
p.myAge()                                          90
p.MyName()                                         "mohammad"
p.MySalary                                         900
```

في المثال السابق قمنا بإنشاء أكثر من بروتكول ، لتبين بأن الكلاس قد ير أكثر من بروتوكل . كما تلاحظ في السطر الأخير قمنا بإشتقاق كائن من الكلاس person ومن ثم قمنا بالوصول للعمليات الاختيارية و الاجبارية ، و هم العمر و الاسم و الراتب.

الخاتمة

في نهاية هذا الكتاب أتمنى لكم التوفيق و التيسير في فهم البرمجة كائنية التوجه ، فهذه المصطلحات و المفاهيم لا تنطبق فقط على لغة swift بل هي شاملة كل اللغات مع اختلاف استدعائها فقط.

كما أتمنى منكم الدعاء لجميع المؤمنين والمؤمنات بالرحمة ، و أن أنال منكم دعوة لوالدتي رحمها الله بأن يثبتها الله عند السؤال و لي و لأهلي.

توجد دورة تكمل لك هذا الموضوع موجودة على الانرنت يمكن الذهاب لها من الرابط التالي ، https://www.youtube.com/watch?v=Yp5AP_4EMmM&list=PLz0bp5hwZAyUygQV-pKC55v4sEtk6JuKw

كما توجد لدي دورة مدفوعة تتكلم عن التعامل مع قواعد البيانات و البرامج الخاصة بالايفون بلغة سويفت على الرابط التالي ، https://www.udemy.com/database-on-web-arabic/

نبذه عن الكاتب

مفيد الشعيب ، ببساطة مبرمج لبرامج الآيفون والآيباد و صاحب أقوى وأول دورة لبرامج الآيفون على اليوتيوب مجانية ، تحتوي على أكثر من ١٠٠ درس و لا تزال في تطور ولن تتوقف ، دخلت المجال عن موهبة فتبحرت فيه. كثيرا ما أردد بأن المستحيل ليس مستحيل وأننا ولدنا كأناس عاديين جدا لذلك يجب أن لا نفضل الموت على هذا النهج ، بل نسير نحو مواهبنا و قدراتنا الكامنة.

حسابي ع الانستقرام @hh288

الفهرس

- العادات الخاطئة في تعلم البرمجة ٣ - ٤ ص
- ماهي لغات البرمجة ٥ ص
- مقدمة عن برمجة كائنية التوجه ٧ ص
- تحميل الاكس كود ٨ ص
- المتغيرات و الثوابت ٩ - ١١ ص
- التعليقات ١٢ ص
- العمليات الحسابية ١٣ ص
- التوبلز ١٤ ص
- المصفوفات ١٥ - ١٦ ص
- القاموس ١٧ ص
- الشروط ١٨ -٢٠ ص
- قاعدة swicth ٢١ ص
- حلقات التكرار ٢٢ - ٢٣ ص
- أوامر القفز ٢٤ - ٢٥ ص
- التعامل مع النصوص ٢٦ ص
- الدوال ٢٧ - ٢٩ ص
- الكلوجرز ٣٠ -٣١ ص
- الـ enum ٣٢ ص
- المتغيرات المحلية و العامة ٣٣ ص
- المباني ٣٤ - ٣٥ ص
- القيم المحسوبة ٣٦ ص
- الخصائص الكسولة ٣٧ ص
- خصائص المراقبة ٣٨ ص
- الخصائص الخاصة بالمبنى ٣٩ ص
- دوال المباني ٤٠ ص
- الدوال و mutating ٤١ ص
- سر كلمة self ٤٢ ص
- الـ supscript ٤٣ ص
- دوال البناء ٤٤ ص
- ما هو الكلاس ٤٥ ص
- القيم المحسوبة ٤٦ ص
- دوال البناء في الكلاس ٤٧ ص

- الدوال الخاصة بالكلاس ٤٨ ص
- الوراثة ٤٩ - ٥٠ ص
- مبدأ التحميل الزائد ٥١ ص
- مبدأ overriding ٥٢ ص
- الكلمة final ٥٣ ص
- التعدد الصوري ٥٤ ص
- دالة الهدم ٥٥ ص
- علامة الاستفهام و التعجب ٥٦ ص
- كلمات weak .... الخ ٥٧ ص
- كلمات any object , any ٥٨ ص
- البروتوكول ٥٩ - ٦٠ ص